A Light-Weight Cooperative Multi-threading with Hardware Supported Thread-Management on an Embedded Multi-Processor System

Bo-Cheng Charles Lai
EE Department
UCLA
CA 90095-1594
bclai@ee.ucla.edu

Patrick Schaumont
ECE Department
Virginia Tech.
VA
schaum@vt.edu

Ingrid Verbauwhede
EE Dept. UCLA, CA
and
ESAT, K.U.Leuven, BE
Ingrid@ee.ucla.edu

ABSTRACT

This paper proposes a light-weight cooperative multi-threading programming model for an embedded multi-processor system. The synchronization between different threads is obtained by a test-and-set-lock. Each processor needs to acquire the lock before accessing shared resources. To reduce the overhead of software thread-management, a hardware thread-queue manager is added to maintain the stack pointer for each thread. This reduces off-chip memory accesses during context switches. The whole multi-processor system, including software stack and hardware architecture, is evaluated with a cycle-accurate simulation platform. With a data-flow-based image encoder as the driver application, the multi-processor system with hardware thread-queue manager achieves 9.5% performance speedup compared to a pure software thread-queue manager.

I. Introduction

In traditional single-processor systems, the approach of increasing clock frequency can no longer bring up required performance enhancement [1]. This is due to the fact that it is becoming difficult and cost-inefficient to expose more instruction level parallelism. A Multi-processor system-on-chip (MPSOC) has been proposed to achieve better performance [2]. Multiple cores on a chip offer higher parallel computation capability and thus potentially higher performance. In order to take advantages out of the parallel computation capability provided by multi-processors, the parallelism of an application has to be exposed by partitioning the application into tasks that can be executed concurrently. Threads are a well-known software abstraction for task-level parallelism [3]. In a multi-processor system, applications are partitioned into threads. By executing threads on multiple cores concurrently, the performance can be enhanced and applications can be completed faster.

In this paper, we propose a light-weight cooperative multi-threading programming model which supports multi-threading execution on a shared-memory SMP (Symmetric Multi-Processing) architecture. By adapting a cooperative multi-threading scheme, all context switches are done with known processor states, and processor registers do not need to be stored. The stack of context information is stored in a circular queue which resides in main memory. Thus every context switch needs to read the pointer of a stack before the system can locate the stack. In order to further speed up context switches, a low-complexity hardware thread-queue manager is proposed to accelerate accessing the stack pointers. With a data-flow-based image encoder as the driver application, the multi-processor system with hardware thread-queue manager achieves up to 9.5% performance speedup compared to a pure software thread-queue manager. The whole multi-processor system, including multi-threading software and SMP hardware architecture, is simulated by a cycle-accurate simulation platform.

This paper is organized as follows. Section II discusses the related work of multi-threading programming model and multi-processor systems. Section III gives brief introduction to the SMP architecture used in this paper. Section IV details the cooperative multi-threading programming model and library. A hardware thread-queue manager which supports accessing stack pointers will be discussed in Section V. Section VI introduces a data-flow image encoder which will be used as the application driver. Simulation results will be shown in Section VII. Section VIII gives the conclusion of this paper and related future work.

1-4244-0132-1/05/$20.00 ©2005 IEEE

II. Related Work

Task-level parallelism is a newcomer in the area of on-chip parallellism. Indeed, over the past years, instruction-level parallelism has received more attention, both academically as well as commercially. This has resulted in commercially available ASIP (such as Xtensa from Tensilica) and VLIW architectures (such as Optimode from ARM). ARM's MPCore [11] proposes a SMP multi-processor system. They support symmetric energy/speed scaling, in which the voltage and frequency of all cores is varied together. In contrast to our approach, they use a heavy kernel-thread programming model, based on Linux SMP.

POSIX thread [8], or Pthread, is a widely used multi-threaded interface that has been specified for UNIX systems. It provides APIs (Application Programming Interfaces) for programmers to create threads and manage the parallel execution of the threads. However, it requires support by a heavy OS-kernel such as Linux SMP, which makes it impractical to be mapped on a resource-limited embedded system. MultiFlex [9] is a multi-processor programming environment, which can support a message passing model and a shared-memory symmetric multi-processing model (SMP). In order to be used in a resource-constrained system, MultiFlex is implemented by a combination of a light-weight software layer and a hardware concurrency engine, which is similar to our system organization.

III. Shared-Memory SMP Architecture

A shared-memory SMP architecture is a common model for multi-processor systems. Each processing core in a SMP architecture is identical such that a thread can be executed by any of the processors. Due to the fact that the memory space is shared, a mechanism is required to implement inter-process synchronization and atomic operations, such as accessing semaphores or mutexes.

Here we introduce a shared-memory SMP architecture which will be used as a platform to demonstrate our multi-threading library. Fig.1 shows an instance of a shared-memory SMP architecture. The system contains four ARM processors, and each processor has its own data cache and instruction cache. Four processors are connected by a central bus, which is based on a master/slave scheme. Each processor is a master and can initiate a read- or write-transaction. The bus allows only one transaction at a time. A hardware test-and-set lock supports inter-process communication and synchronization of the system. Processors need to acquire the lock to access shared resources, such as main memory. The main memory resides in off-chip memory which can be accessed through the memory interface. Due to the cooperative multi-threading model, all the context switches are done with known processor states. During a context switch, a processor needs to store the stack pointer of the current running thread and to load a stack pointer of the next thread. A hardware thread-queue manager is added to reduce the overhead of accessing off-chip memory for stack pointers during context switches.

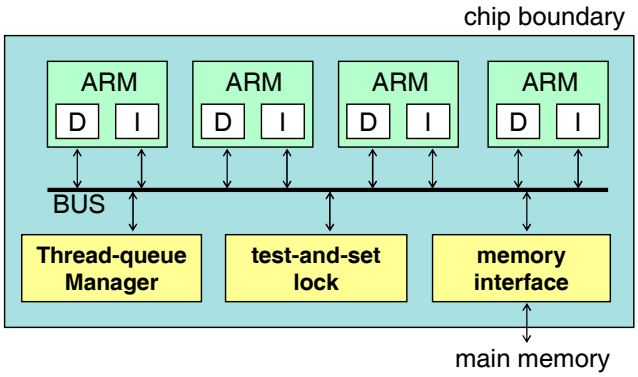

Fig.1: Multi-processor architecture with hardware thread-queue manager and test-and-set lock

A cycle-accurate simulation platform [10] has been implemented to simulate the whole SMP system and return cycle-accurate performance. The processor core is modeled by SimIt-ARM[4], an instruction set simulator of the StrongARM micro-architecture. The other hardware modules, including the thread-queue manager, the test-and-set lock and the memory interface, are modeled and integrated by GEZEL[5], which is a cycle-accurate software/hardware co-simulation environment.

Table 1: Interfaces and models used in the multi-processor simulation platform

	Interfaces and system modules	Tools and models
Software Interfaces	Applications	C/C++
	Multi-threading library	Multi-Processor version of QuickThreads [6][10] (2KB of code)
Hardware Modules	ARM processor	SimIt-ARM ISS [4]
	HW modules, system integration	GEZEL [5]

IV. Cooperative Multi-threading Programming Model

We have implemented a multi-processor version of the QuickThreads cooperative multi-threading library [6]. This programming model has two important advantages. First, it is light-weight. The complete multi-threading library

including boot code fits in under 2Kbytes, which makes it appropriate for resource-limited embedded systems. The second advantage is that it can be implemented by using a standard software tool-chain, which is familiar to software programmers. The multi-threaded program is written in C and compiled using an arm-linux gcc 3.2.2 cross compiler, and thread-management is supported by a software library.

A. *Booting Sequence*

While standard C is used to program the multi-processor, there are at any moment as many threads of control in the program as there are active processors. When the multi-processor starts, the boot code may allow only a single processor to enter `main()`. This is solved, as illustrated in Fig.2, by means of a boot lock L_b. When the processors execute the boot code, they will compete for this lock. The processor that grabs the lock executes the standard C library initialization code resulting in a call to `main()`. For the other processors, a custom stack frame is created by the boot code and the function `slave_main()` is called instead, with a processor id as argument. The C programmer thus sees a unique entry point for each processor in the application program. Once the application program is running, additional user threads can be created by means of cooperative multi-threading.

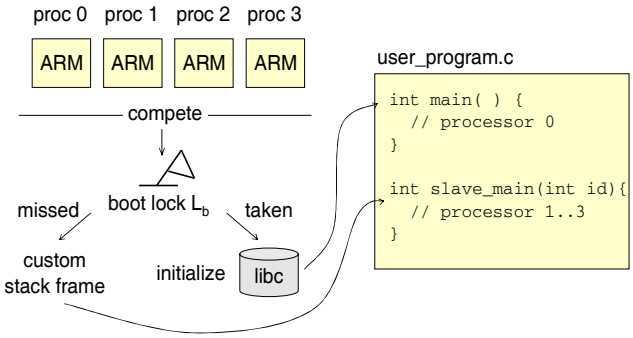

Fig.2: Booting process of a multi-processor system

B. *Cooperative Multi-threading*

Test-and-set lock. The synchronization of the multi-processor system is supported by a hardware test-and-set lock (Fig.3). Two atomic operations, `tsread()` and `tswrite(V)`, are implemented using the lock. The first operation sets the lock to TRUE and returns the previous value. The second operation writes value V to the lock. The semaphores and mutexes can be implemented by using these two operations by means of spin-lock [7]. When a processor wants to acquire the hardware lock, it will repeatedly check `tsread()` until it returns FALSE. The lock can be released by calling `tswrite(FALSE)`.

As illustrated in Fig.3, additional test-and-set locks can be implemented in software on top of the hardware lock. In that case, the hardware lock is used to protect a memory location that holds the value of a software lock. This way, the amount of dedicated hardware for synchronization is kept minimal.

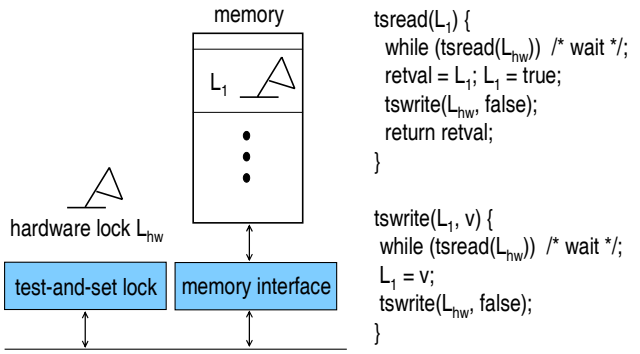

Fig.3: Test-and-set lock and `tsread()/tswrite()`

Thread queue. Context information of threads is stored in a circular queue (Fig.4). The context information contains a thread stack and a stack pointer. All processors use the thread queue for context switches. The queue is protected by a test-and-set lock Lq to ensure the thread queue can be modified by a single processor at a time. Threads can be constructed and entered the thread queue with `create()`. When a processor calls `start()`, it will switch from the main thread to the first thread in the queue. The `yield()` function puts the thread back into the thread queue, and the `abort()` function can terminate a thread.

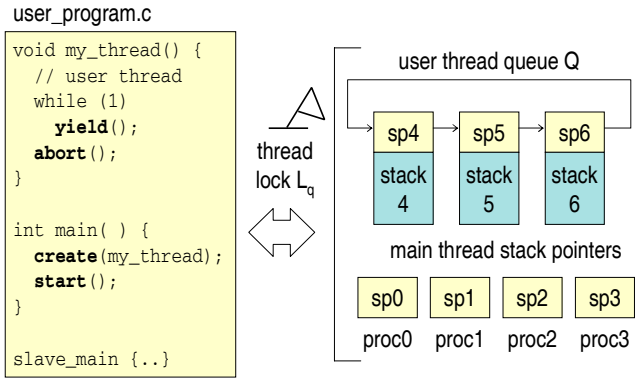

Fig.4: Thread programming model and thread queue

In the software thread library, the thread queue is located in the main memory. Therefore each context switch requires access to the main memory for context information. Accessing main memory, which is usually implemented as an off-chip memory, impairs the system performance due to long memory-access latencies. A hardware thread-queue

manager, which will be further discussed in Section V, is added to speed up the context switch by keeping stack pointers within a chip.

C. An Example: Accumulating Numbers

Here we use a simple example to illustrate the multi-threaded programming interface for task level parallellization on C programming. We wish to accumulate a list of one million numbers stored in off-chip memory. Listing 1 shows a version of the code that uses 4 threads. Each thread accumulates 250K numbers with accumulate (lines 2—10). Thread completion is signaled with a lock-protected counter (lines 7—9). In the main function, the four threads are created (lines 13—16) and executed through completion (lines 18—19 and 26—27). The execution cycles of this example on a single-processor and a quad-processor system are 8.0M and 2.1M respectively. It clearly shows that the task-level parallelization really takes advantages of the parallel computation capability provided by a multi-processor system.

LISTING 1. Accumulating Numbers Example

```
1.  int thread_done = 0;
2.  void accumulate(long long *r) {
3.    long long acc = 0;
4.    for (int i=0; i < 250000; i++)
5.    acc += numbers[i + 250000*thread_id()];
6.    *r = acc;
7.    while (tsread(DONE_LOCK));
8.    thread_done++;
9.    tswrite(DONE_LOCK,0);
10. }
11. int main(int argc, char **argv) {
12.   long long result[4];
13.   create(accumulate, &(result[0]));
14.   create(accumulate, &(result[1]));
15.   create(accumulate, &(result[2]));
16.   create(accumulate, &(result[3]));
17.
18.   while (thread_done < 4)
19.     start();
20.   for (int i=1; i<4; i++)
21.     result[i] += result[i-1];
22.   exit(0);
23. }
24. void slave_main(int procid) {
25.
26.   while (thread_done < 4)
27.     start();
28.
29.   exit(0);
30. }
```

V. Hardware Support Thread Queue Management

Fig.5 illustrates the architecture of a thread-queue manager. It is composed of a register file and a controller. The manager is connected to the central bus. Because it is a circular queue, two indexes, q_head and q_tail, need to be maintained. When a thread is created or yielded, the stack pointer of the thread will be stored into the thread-queue manager. When a processor is idle or needs to context switch to a different thread, it will ask the thread-queue manager for the stack pointer of the next thread. The access to the thread-queue manager is merged with the thread library, thus the atomicity is ensured by the test-and-set lock. The controller only needs to store/load the stack pointer to/from the register file, maintain the q_head/q_tail of the stack pointer queue, and implement the bus protocol. This results in low hardware cost and complexity for the thread-queue manager.

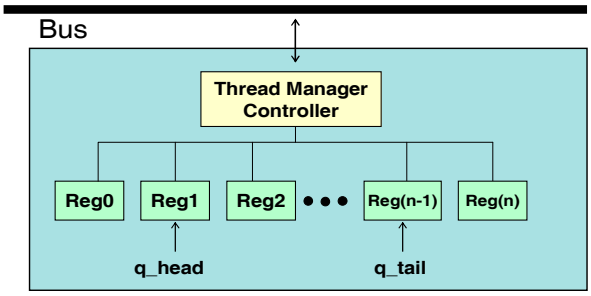

Fig.5: Thread-queue manager

VI. Data-flow Image Encoder

A data-flow image encoder is used as the application driver. Data-flow model is commonly used in digital signal processing applications. It is similar to a producer/consumer model. The system is composed of actors, where each of them represents a functional block of the system and can be executed concurrently. After processing the input data, actors produce output data, which will be input into the subsequent actors. Fig.6 shows an image encoder which is implemented as a data-flow system. The circles represent the actors, and the bars represent the intermediate queues which are used for passing data between actors. When mapping onto a multi-processor system, each actor is implemented as a thread and can be executed by any processor cores in the system. Intermediate queues are located in the main memory of a multi-processor system. Thus every access to the queues is translated as traffic on the bus to access the external memory. There are a total of 26 actors in the system.

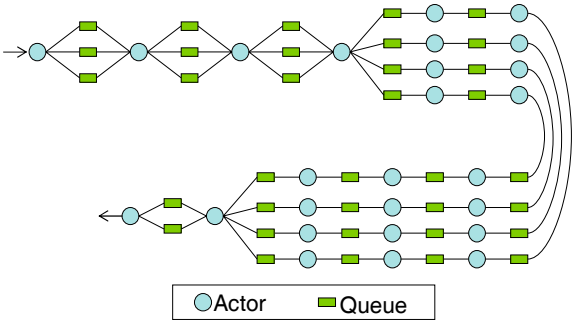

Fig.6: Data-flow image encoder

VII. Results

Fig.7 shows the performance of different multi-processor systems. The results are based on cycle-accurate simulations. The increasing number of processors in a system will enhance the overall performance. For systems with pure software thread-management, the dual-processor and the tri-processor systems are 29% and 28% faster than the single-processor system respectively. However, more synchronization traffic is induced when the number of processors is increased. Also, contention for the central bus impairs the overall system performance. The quad-processor system is only 16% faster than the single-processor system. It is even slower than the dual- and tri-processor system.

The hardware thread-queue manager maintains the stack pointer information on chip, thus processors do not need to access the external memory for stack pointers during context switches. Compared to the pure software thread library, multi-processor systems with hardware thread-queue manager give 3.7% to 9.5% performance enhancement.

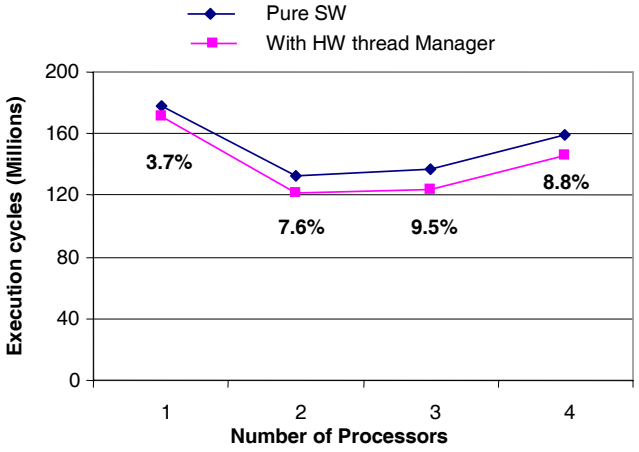

Fig.7: Performance of multi-processor systems

VIII. Conclusion

This paper proposed a cooperative multi-threading library and programming model on an embedded multi-processor system. The programming model is based on a light-weight cooperative multi-threading library, which fits in 2KB of object codes. A low complexity hardware thread-queue manager is added to reduce the stack pointer accesses to the external memory during context switches. Demonstrated by cycle-accurate simulation results, the hardware thread-queue manager offers 3.7% to 9.5% performance enhancement.

We are currently working on more hardware extensions to support multi-threading on embedded multi-processors. We are also implementing an energy-efficient multi-processor platform, which adapts energy scaling to achieve energy reduction. Next to a data-flow model, other multi-threading models are being evaluated as well.

Acknowledgement

The authors gratefully acknowledge the support of NSF (Grant CCR 0310527) and SRC (Grant SRC-2003-HJ-1116).

References

[1] A. Jerraya, W. Wolf, “Multi-processor Systems-on-Chips,” Morgan Kaufmann, Sept 2004, ISBN 0-12-385251-X.

[2] L.Hammond, B.A.Nayfeh, K.Olukotun, “A Single-Chip Multiprocessor,” *Proc. of IEEE*, pp.79-85, Sept. 1997.

[3] D.E. Culler, J.P. Singh, A. Gupta, “Parallel Computer Architectures: A Hardware/Software Approach,” MKP Publishers, 1999, ISBN 1-55860-343-3

[4] W. Qin, S. Malik. Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation, 2003 Design Automation and Test in Europe, pp.556-561, March 2003.

[5] P. Schaumont, I. Verbauwhede, "Interactive cosimulation with partial evaluation," 2004 Design Automation and Test in Europe, pp.642-647, February 2004.

[6] D. Keppel, “Tools and Techniques for Building Fast Portable Threads Packages,” UWCSE 93-05-06, U. Washington, 1993.

[7] G. Andrews, “Concurrent programming - principles and practice”, pp.102-105, Benjamin Cummings Publ. 1991.

[8] B. Nichols, D. Buttlar, J.P. Farrell, J. Farrell, “Pthreads Programming: A POSIX Standard for Better Multiprocessing,” O'Reilly Publisher, 1996, ISBN 1-56592-115-1

[9] P.G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, G. Nicolescu, “Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management,” CODES+ISSS, pp.48-53, Sept. 2004.

[10] P. Schaumont, B.C. Lai, W. Qin, I. Verbauwhede, "Cooperative multi-threading on embedded multi-processor architectures enables energy-scalable design,” Proc. 2005 Design Automation Conference, pp. 27-30, June 2005.

[11] J. Goodacre, A. Sloss, “Parallelism and the ARM Instruction Set Architecture,” IEEE Computer, vol.38, no.7, pp.42-50, July 2005.